A Hoare Calculus for the Verification of Synchronous Languages

Manuel Gesell, Klaus Schneider
http://es.cs.uni-kl.de

Embedded Systems Group
University of Kaiserslautern

International Open Workshop on Synchronous Programming
2011

Table of Contents

Outline

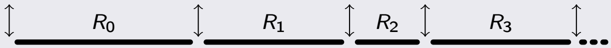
What is this Talk about?

- synchronous languages
- Quartz
- Hoare calculus
- synchronous tuple assignment form
- a Hoare calculus for Quartz

Synchronous Model of Computation

- abstract time to sequence of reactions (instants)
- each variable has one value per instant
- inputs and outputs are read and produced for an instant
- coincides with clock-cycles of synchronous circuits
  - gate delays mimic computation
  - one value per wire for each clock cycle
- instants are a logical time-scale

Synchronous Trace

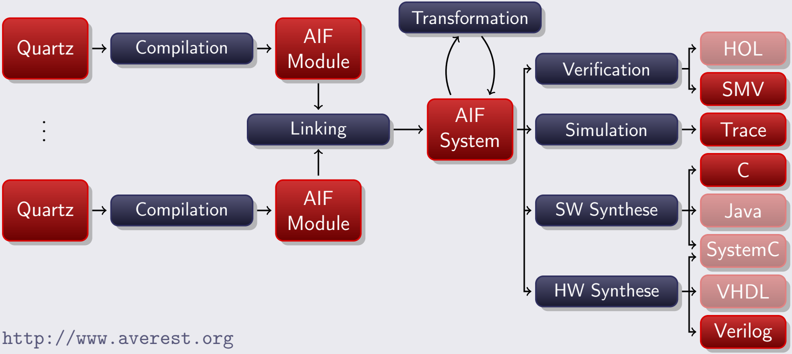


Synchronous Languages

- implement the synchronous model
- can be used for hardware and software
- data-flow oriented languages
  - Lustre
  - Signal
- control-flow oriented languages (imperative)
  - Quartz
    - developed in our working group
    - Averest toolset
  - Esterel

Averest

Averest Design Flow



http://www.averest.org

Outline

Outline

Quartz Example

```
module P1 (nat ?i1,?i2,o1,o2)
{
  nat x;
  loop {
    o1 = i1 + i2;
    x  = i1;
    pause;
    o1 = o2 + i1 + x;
    o2 = i2;
    x  = 2;
    pause;
    if (i1 > 4)
      o1 = i1;
    o2 = i1 + o1;
    pause;
  }
}
```

- **pause** marks end of a step
- i1, i2 are inputs, o1, o2 are outputs, x is a local variable

	1	2	3	4	5
i1	1	2	3	4	5
i2	2	4	6	8	0
x	1	2	2	4	2
o1	3	8	8	12	7
o2	0	4	11	11	0

Quartz Statements

- assignments: x=α, **next**(x)=α
- end of step: **pause**
- conditional execution: **if**(γ)... **else** ...
- loops: **while**(γ){ ... }, **loop**{ ... }
- abortion: **abort** ... **when**(γ)
  - various variants
  - aborts execution when condition γ holds
- suspension: **suspend** ... **when**(γ)
  - various variants
  - suspens execution when condition γ holds
- concurrent execution: { ... } || { ... }
- ...

Causal Dependencies

```
module P2(bool o)
{
  bool x;
  o = x;
  x = true;
}
```

- value for o holds for the whole step
- both actions are executed according to their data dependencies
- P2 is causally correct in sense of Quartz

```
module P3(bool o)
{
  if(!o)
    pause;
  o = true;
}
```

- if o=**true** is reached depends on o
- o = **true** would lead to a valid execution of the program
- P3 is not causally correct in sense of Quartz

Outline

Why Hoare...

model checking

- fully automatic
- suffers from state-space explosion problem
- enumerates all possible values

interactive verification based on Hoare calculus

- interactive (semi-automatic)
- requires additional invariants
- allows abstraction from the size of data structures as well as the data-types itself

An integration of model checking and interactive verification is desired.

Hoare Calculus

nothing : $$\frac{}{\{\Phi\}\ \texttt{nothing}\ \{\Phi\}}$$

assign : $$\frac{}{\{[\Phi]_x^\tau\}\ x = \tau\ \{\Phi\}}$$

sequence : $$\frac{\{\Phi_1\}\ S_1\ \{\Phi_2\} \quad \{\Phi_2\}\ S_2\ \{\Phi_3\}}{\{\Phi_1\}\ S_1; S_2\ \{\Phi_3\}}$$

conditional : $$\frac{\{\sigma \wedge \Phi\}\ S_1\ \{\Psi\} \quad \{\neg\sigma \wedge \Phi\}\ S_2\ \{\Psi\}}{\{\Phi\}\ \texttt{if}(\sigma)\ S_1\ \texttt{else}\ S_2\ \{\Psi\}}$$

loop : $$\frac{\{\sigma \wedge \Phi\}\ S\ \{\Phi\}}{\{\Phi\}\ \texttt{while}(\sigma)\ S\ \{\neg\sigma \wedge \Phi\}}$$

weaken : $$\frac{\models \Phi_1 \rightarrow \Phi_2 \quad \{\Phi_2\}\ S\ \{\Phi_3\} \quad \models \Phi_3 \rightarrow \Phi_4}{\{\Phi_1\}\ S\ \{\Phi_4\}}$$

Hoare Calculus

nothing : $$\frac{}{\{\Phi\}\ \texttt{nothing}\ \{\Phi\}}$$

assign : $$\frac{}{\{[\Phi]_x^\tau\}\ x = \tau\ \{\Phi\}}$$

sequence : $$\frac{\{\Phi_1\}\ S_1\ \{\Phi_2\} \quad \{\Phi_2\}\ S_2\ \{\Phi_3\}}{\{\Phi_1\}\ S_1; S_2\ \{\Phi_3\}}$$

conditional : $$\frac{\{\sigma \wedge \Phi\}\ S_1\ \{\Psi\} \quad \{\neg\sigma \wedge \Phi\}\ S_2\ \{\Psi\}}{\{\Phi\}\ \texttt{if}(\sigma)\ S_1\ \texttt{else}\ S_2\ \{\Psi\}}$$

loop : $$\frac{\{\sigma \wedge \Phi\}\ S\ \{\Phi\}}{\{\Phi\}\ \texttt{while}(\sigma)\ S\ \{\neg\sigma \wedge \Phi\}}$$

weaken : $$\frac{\models \Phi_1 \rightarrow \Phi_2 \quad \{\Phi_2\}\ S\ \{\Phi_3\} \quad \models \Phi_3 \rightarrow \Phi_4}{\{\Phi_1\}\ S\ \{\Phi_4\}}$$

A Hoare calculus for Quartz

- defining a Hoare calculus only requires the definition of a Hoare rule for each statement
- it is possible to synthesis a Quartz program to sequential code and then apply the classical Hoare calculus

Outline

Problems Defining a Hoare Calculus for Quartz

problems defining a Hoare calculus for Quartz on statement level

- inputs are read in each macro step

  $\Rightarrow$ reaching a **pause**: update inputs and depended conditions
- each statement rule requires to regard many cases
- macro step must be identified
- all variable updates have to be done synchronously
- in case no assignment is done the default value must be used

Problems Defining a Hoare Calculus - Many Cases

$\{\Phi\}\ S_1; S_2\ \{\Psi\} \Rightarrow$

- case: $\mathrm{inst}(S_1) \wedge \mathrm{inst}(S_2)$
- case: $\neg\mathrm{inst}(S_1) \wedge \mathrm{inst}(S_2)$
- case: $\mathrm{inst}(S_1) \wedge \neg\mathrm{inst}(S_2)$
- case: $\neg\mathrm{inst}(S_1) \wedge \neg\mathrm{inst}(S_2)$

even worse: $\{\Phi\}\ S_1 || S_2\ \{\Psi\}$

Problems Defining a Hoare Calculus for Quartz

problems defining a Hoare calculus for Quartz on statement level

- inputs are read in each macro step

  $\Rightarrow$ reaching a **`pause`**: update inputs and depended conditions
- each statement rule requires to regard many cases
- macro step must be identified
- all variable updates have to be done synchronously
- in case no assignment is done the default value must be used

Macro Step Behaviour

Each variable in a macro step has a unique value. Either determined by

- an delayed assignment in the previous step,
- an immediate assignment in the current step or
- a type dependent default value

Macro Step Behaviour

P4

```
if (a) {
 x = 5;
 y = true;
} else {
 y = false;
}

pause;
```

P5

```
if (a) {
 x = 5;
 y = true;
} else {
 y = false;
}
if (!y) x = 3;
pause;
```

P6

```
if (a) {
 x=5;
 y = true;
} else {
 y = false;
}
if (!y & b) x = 3;
pause;
```

Macro Step Behaviour

P4

```
if (a) {
 x = 5;
 y = true;
} else {
 y = false;
}

pause;
```

P5

```
if (a) {
 x = 5;
 y = true;
} else {
 y = false;
}
if (!y) x = 3;
pause;
```

P6

```
if (a) {
 x=5;
 y = true;
} else {
 y = false;
}
if (!y & b) x = 3;
pause;
```

$!a \Rightarrow x = 0$

Macro Step Behaviour

P4

```
if (a) {
 x = 5;
 y = true;
} else {
 y = false;
}

pause;
```

P5

```
if (a) {
 x = 5;
 y = true;
} else {
 y = false;
}
if (!y) x = 3;
pause;
```

P6

```
if (a) {
 x=5;
 y = true;
} else {
 y = false;
}
if (!y & b) x = 3;
pause;
```

`!a`$\Rightarrow$`x = 0`

no default value required

`!a`$\rightarrow$`!y`

Macro Step Behaviour

P4

```
if (a) {
 x = 5;
 y = true;
} else {
 y = false;
}

pause;
```

P5

```
if (a) {
 x = 5;
 y = true;
} else {
 y = false;
}
if (!y) x = 3;
pause;
```

P6

```
if (a) {
 x=5;
 y = true;
} else {
 y = false;
}
if (!y & b) x = 3;
pause;
```

!a⇒x = 0

no default value required

!a & !b⇒x = 0

!a→!y

Hoare Calculus

P7

```
if (a) {
   ...
   pause;
} else {
   x = 3;
}

if (x >= 2)
   ...

if (a) x = 7;
pause;
```

- variable's default value may be read!
- determine the necessary of the default value cannot be done locally (on statement level)

Hoare Calculus

P7

```
if (a) {
    ...
    pause;
} else {
    x = 3;
}

if (x >= 2)
    ...

if (a) x = 7;
pause;
```

- variable's default value may be read!
- determine the necessary of the default value cannot be done locally (on statement level)

Hoare Calculus

P7

```
if (a) {
    ...
    pause;
} else {
    x = 3;
}

if (x >= 2)
    ...

if (a) x = 7;
pause;
```

- variable's default value may be read!
- determine the necessary of the default value cannot be done locally (on statement level)

Hoare Calculus

P7

```
if (a) {
    ...
    pause;
} else {
    x = 3;
}

if (x >= 2)
    ...

if (a) x = 7;
pause;
```

!a

a

- variable's default value may be read!
- determine the necessary of the default value cannot be done locally (on statement level)

Definition of a Hoare Calculus for Quartz

- define two-stage Hoare-like rules.
  1. identify macro step
  2. reason about the macro step

Definition of a Hoare Calculus for Quartz

- define two-stage Hoare-like rules.
  1. identify macro step
  2. reason about the macro step
- split the verification process into these stages.
  1. source-code transformation that collects all macro step's actions
  2. reason about code in a certain normal form

Synchronous Tuple Assignments (STA)

collecting all actions in synchronous tuple assignments (STAs)

Definition (Synchronous Tuple Assignment (STA))

Given that $x_1 = \tau_1$, ..., $x_m = \tau_m$ and $\mathbf{next}(y_1) = \pi_1$, ..., $\mathbf{next}(y_m) = \pi_m$ are assignments with pairwise different left-hand side expressions x_i and y_i, and given that these assignments are causally ordered such that there are no read-after-write conflicts, i. e. that τ_i only has occurrences of x_1, ..., x_{i-1}, then we call the following statement a synchronous tuple assignment:

$$(x_1, \ldots, x_m).(y_1, \ldots, y_n) = (\tau_1, \ldots, \tau_m).(\pi_1, \ldots, \pi_n)$$

Quartz Programs in STA Form

Definition (Quartz Programs in STA Form)

A Quartz program is in synchronous tuple assignment (STA) form if all its actions are STAs and between the execution of two STAs at least one **`pause`** is executed.

Fibonacci Numbers

```
module Fib(nat ?i,f,event !r)

{
  nat k,g,n;
  n = i;
  if(n <= 0)
    f=0;
  else {
    k = 1;
    g = 0;
    f = 1;
    while(k != n) {
      next(g) = f;
      next(f) = f+g;
      next(k) = k+1;
      l: pause;
    }
  }
  emit(r);
}
```

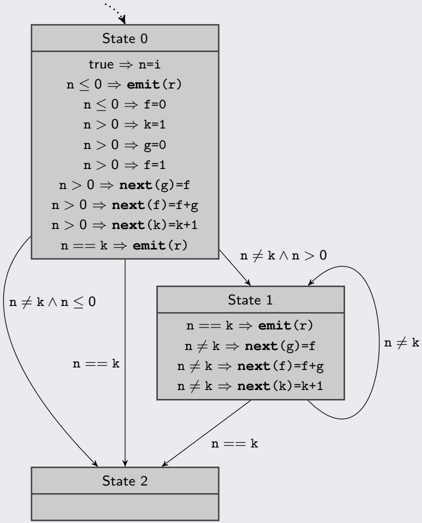
- computes Fibonacci numbers in quartz
- $\mathtt{r} \rightarrow \mathtt{f == FIB (i_0)}$

Fibonacci Numbers

```
module Fib(nat ?i,f,event !r)

{
  nat k,g,n;
  n = i;
  if(n <= 0)
    f=0;
  else {
    k = 1;
    g = 0;
    f = 1;
    while(k != n) {
      next(g) = f;
      next(f) = f+g;
      next(k) = k+1;
      l: pause;
    }
  }
  emit(r);
}
```

EFSM for Modul Fib



Fib in STA form (automatic-version)

module FSA(nat ?i,f,event r)

```
{
  nat k,g,n,l;
  do {
    case
      (l==0) do //State 0
        (n,r,k,g,f).(g,f,k,l) =
          (i,n<=0,1,0,(n>0?1:0)).
          (f,f+g,k+1,(n>0&n!=k?1:2));
      (l==1) do //State 1
        (r).(g,f,k,l) =
          (n==k).
          (f,f+g,k+1,(n!=k?1:2));
    default
      nothing;
    pause;
  } while (l!=2);
}
```

- structure completely destroyed
- code contains only a single loop
- same drawbacks as synthesising sequential code

Fib in STA form (handwritten-version)

```
module Fib(nat ?i,f,event !r)

{
  nat k,g,n;
  n = i;
  if(n <= 0)
    f=0;
  else {
    k = 1;
    g = 0;
    f = 1;
    while(k != n) {
      next(g) = f;
      next(f) = f+g;
      next(k) = k+1;
      l: pause;
    }
  }
  emit(r);
}
```

```
module FSH(nat ?i,f,event !r)

{
  nat k,g,n;
  if(n<=0) {
    (n,f,r).() = (i,0,true).();
  } else {
    (n,k,g,f,r).(g,f,k) =
      (i,1,0,1,k==n).(f,f+g,k+1);
    while(k!=n) {
      pause;
      (r).(g,f,k) = (k==n).
          (f,f+g,k+1);
    }
  }
}
```

Fib in STA form (handwritten-version)

```
module FSH(nat ?i,f,event !r)

{
  nat k,g,n;
  if(n<=0) {
    (n,f,r).() = (i,0,true).();
  } else {
    (n,k,g,f,r).(g,f,k) =
      (i,1,0,1,k==n).(f,f+g,k+1);
    while(k!=n) {
      pause;
      (r).(g,f,k) = (k==n).
          (f,f+g,k+1);
    }
  }
}
```

- structure is preserved
- assignment are shifted and/or duplicated
- same invariants are usable

Required Rules

STA Rule

$$\frac{}{\{\left[\ldots\left[[\Phi]^{\pi_1,\ldots,\pi_n}_{y'_1,\ldots,y'_n}\right]^{\tau_n}_{x_n}\ldots\right]^{\tau_1}_{x_1}\}\ (x_1,\ldots,x_m).(y_1,\ldots,y_n) = (\tau_1,\ldots,\tau_m).(\pi_1,\ldots,\pi_n)\ \{\Phi\}}$$

Pause Rule

$$\frac{}{\{\left[[\ldots\Phi\ldots]^{\tau_1\ldots\tau_n}_{i_1,\ldots i_n}\right]^{y'_1\ldots y'_n}_{y_1\ldots y_n}\}\ \mathbf{pause}\ \{\Phi\}}$$

STA form

STA form is reasonable

- all Quartz programs representable
- Hoare rules are easily adaptable
- code structure is preservable

Outline

Summary

- discussed issues defining Hoare rules for Quartz
- introduced a new kind of normal form for Quartz
- showed the practicability of STA form
- extended the set of Hoare rules for Quartz

What needs to be done?

- further work: defining a structure preserving transformation (partially done)
- alternative idea: defining rules on AIF level, but user provides invariants and chose rules on source-code level. Advantages are:
    - reuse of schizophrenia and causality techniques
    - AIF transformations are apply-able before verification
    - no need for STA transformation (implicitly usage)
    - invariants of source code are usable
    - no need to verify the compiling procedure

thank you for your attention